PSYNC: A partially synchronous language for fault-tolerant distributed algorithms

Cezara Drăgoi, Thomas Henzinger, Damien Zufferey

PSYNC: A partially synchronous language for fault-tolerant distributed algorithms

Cezara Drăgoi
INRIA, ENS, CNRS
cezara.dragoi@inria.fr

Thomas A. Henzinger *
IST Austria
tah@ist.ac.at

Damien Zufferey †
MIT CSAIL
zufferey@csail.mit.edu

Abstract

Fault-tolerant distributed algorithms play an important role in many critical/high-availability applications. These algorithms are notoriously difficult to implement correctly, due to asynchronous communication and the occurrence of faults, such as the network dropping messages or computers crashing.

We introduce PSYNC, a domain specific language based on the *Heard-Of* model, which views asynchronous faulty systems as synchronous ones with an adversarial environment that simulates asynchrony and faults by dropping messages. We define a runtime system for PSYNC that efficiently executes on asynchronous networks. We formalise the relation between the runtime system and PSYNC in terms of observational refinement. This high-level synchronous abstraction introduced by PSYNC simplifies the design and implementation of fault-tolerant distributed algorithms and enables automated formal verification.

We have implemented an embedding of PSYNC in the SCALA programming language with a runtime system for partially synchronous networks. We show the applicability of PSYNC by implementing several important fault-tolerant distributed algorithms and we compare the implementation of consensus algorithms in PSYNC against implementations in other languages in terms of code size, runtime efficiency, and verification.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Verification

Keywords Fault-tolerant distributed algorithms, Round model, Partially synchrony, Automated verification, Consensus

1. Introduction

The need for highly available data storage systems and for higher processing power has led to the development of distributed systems. A distributed system is a set of independent nodes in a network, that communicate and synchronize via message passing, giving the illusion of acting as a single system. The difficulty in designing these systems comes from the network unreliability and the host failures: messages can be dropped and nodes can crash. The processes have only a limited view over the entire system and they must coordinate to achieve global goals.

A general mechanism for implementing a fault-tolerant service is replication: the application is copied on different replicas which are kept consistent. Clients send requests to a replica running the service, the service communicates with the other nodes to maintain a consistent state of the global system, and replies to the clients. Consistency is maintained by solving *consensus problems*. Each process has an initial value, and all processes have to agree on a unique decision value among the initial ones. Therefore all replicas return the same value when queried for the attribute of an object. Consensus algorithms have received a lot of attention in academia and industry, because they are at the core of most high-availability systems, but are difficult to design and implement. Because consensus is not solvable in asynchronous networks in the presence of faults [35], a large number of algorithms have been developed [20, 31, 44, 47, 53, 59], each of them solving consensus under different assumptions on the type of faults, and the degree of synchrony of the system. Moreover, many other problems in distributed systems can be reduced to agreement, for example, atomic broadcast has been shown equivalent with consensus. Noteworthy examples of applications from industry that use consensus algorithms include the Chubby lock service [18], which uses the Paxos [47] consensus algorithm, Apache Zookeeper which has a dedicated algorithm [44] for primary-backup replication.

In this paper we unify the modeling, programming, and verification of fault-tolerant distributed algorithms with a domain specific language, called PSYNC, that

- has a high-level, round-based, control structure that helps the programmer focus on the algorithmic questions rather than spending time fiddling with low-level network and timer code;
- compiles into working, efficient systems that preserve the important user-defined properties of high-level algorithms;
- is amenable to automated verification.

Despite the importance of fault-tolerant algorithms, no widely accepted programming model has emerged for these systems. The algorithms are often published with English descriptions, or, in the best case, pseudo-code. Moreover, fault-tolerant algorithms are rarely implemented as published, but modified to fit the constraints and requirements of the system in which they are incorporated [21]. General purpose programming languages lack the primitives to implement these algorithms in a simple way. A developer is forced to choose between low-level libraries, e.g., POSIX, to write network

* Check conclusions This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM) and by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award).

† ToDo ...

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

and timer code, or high-level abstractions, like supervision trees, that come with their own limitations. The first approach leads to programs with a very complex control structure that hides the algorithm's principles. The later may not work because the abstraction's assumptions do not fit the system's requirements.

The complexity of fault-tolerant implementations makes them prone to error so they are prime candidates for automated verification. Their complexity is twofold: (1) the algorithms these implementations are based on, have an intricate data-flow and (2) the implementations have complex concurrent control structure, e.g., an unbounded number of replicas communicate via event-handlers, and complex data structures, e.g., unbounded buffers. Although fault-tolerant algorithms are at the core of critical applications, there are no automated verification techniques that can deal with the complexity of an implementation of an algorithm like Paxos.

The standard programming paradigm for implementing fault-tolerant distributed algorithms requires reasoning about asynchrony and faults separately. Programming languages provide access only to asynchronous communication primitives. A fundamental result on distributed algorithms [35] shows that it is impossible to reach consensus in asynchronous systems where at least one process might crash. Therefore the algorithms that solve consensus make assumptions on the network finer than asynchrony, i.e., they need to reason about time explicitly. In order to reconcile the modeling of various assumptions on network, the algorithms community introduced computational models that uniformly model (a)synchrony and faults using an adversarial environment that drops messages [24, 37]. We take a programming language perspective on this matter and propose a domain specific language, PSYNC, that offers the programmer the illusion of synchrony, uses the adversarial environment to reason about faults and asynchrony, and efficiently executes on asynchronous networks.

High-level computational model. PSYNC is based on the *Heard-Of* model [24], which structures algorithms in communication-closed rounds [32]. A PSYNC program is defined by a sequence of rounds, and has a lockstep semantics where all the processes execute the same round. Each round consists of two consecutive operations: (1) a *send* method for sending messages and (2) an *update* method that updates the local state according to the messages received during the round. A round is communication-closed if all the messages are either delivered in the round they are sent or dropped. Communication-closed rounds provide a clear scope for the messages and the associated computations.

In the HO-model, a distributed system is a set of processes together with an adversarial environment, where the environment determines the set of messages received in a round. Each process has a *Heard-Of set*, denoted HO, which is a variable over sets of process identities exclusively under the control of the environment. HO-sets abstract the asynchronous and faulty behaviors of the network. In a given round, process p receives a message from process q if q sends a message to p and $q \in \mathsf{HO}(p)$. The network's degree of synchrony and the type of faults correspond to assumptions on how the environment picks the HO-sets.

The round structure together with the HO-sets define a notion of abstract time. Using communication-closed rounds, PSYNC introduces a high-level control structure that allows the programmer to focus on the data computation performed by each process to ensure progress towards solving the considered problem, as opposed to spending time on the programming language constructs, e.g., incrementing message counters, setting timers.

Runtime. We have implemented PSYNC as an embedding in Scala. PSYNC programs link against a runtime system that manages the interface with the network and the resources used by the PSYNC program. The code runs on top of an event-driven framework for asynchronous network applications, uses UDP to transmit data, in a context where processes may permanently crash. The main challenge in deriving asynchronous code from a PSYNC program is defining a procedure that decides when to go to the next round while allowing sufficiently many messages to be delivered. Our approached is based on timeouts: roughly during an execution the time spend by a process to receive messages is divided into chunks of equal length, i.e., the timeout. Each chunk corresponds to one round, identifying uniquely set of messages received by the process within that round. The timeout is a parameter that influences performance and can be fine-tuned to take advantage of the network and the algorithm specificity.

We show that for any PSYNC program $\mathcal{P}$ the asynchronous executions of $\mathcal{P}$ generated by the runtime are *indistinguishable* from the lock-step executions of $\mathcal{P}$. Indistinguishability is a relation between executions which roughly says that no process can locally distinguish between them. Intuitively, this relation is important because it is not possible to solve a problem that requires different answers for executions which are indistinguishable from the local perspective of processes.

Using indistinguishability we show that the asynchronous system defined by the runtime of a program $\mathcal{P}$ observationally refines the synchronous system defined by $\mathcal{P}$, that is every behavior of a client that uses the runtime can be reproduced if the client uses $\mathcal{P}$ instead, provided that the client operations are commutative.

Verification. Due to the complexity distributed systems have reached, we believe it is no longer realistic nor efficient to assume that high level specifications can be proved when development and verification are two disconnected steps in the software production process. PSYNC provides a simple round structure whose the lockstep semantics leads to a smaller number of interleavings and simpler inductive characterizations of the set of reachable states. We have identified a large class of specifications which if met by the lock-step semantics of PSYNC then they are also met by its asynchronous semantics used at runtime.

We have implemented a state-based verification engine for PSYNC, that checks safety and liveness properties. The engine assumes the program annotated with inductive invariants and ranking functions, and proves the validity of these annotations and the fact that they imply the specification. In general annotating a program with inductive invariants is a hard task even for an expert. The advantage of PSYNC lies in the simplicity of the required inductive invariants. Compared with asynchronous programming models, the round structure allows looking at the system's invariants at the boundary between rounds and ignores messages.

Contributions. We introduce PSYNC a domain specific language for fault-tolerant algorithms.

- PSYNC makes it possible to write, execute, and verify high-level implementations of fault-tolerant algorithms.
- PSYNC has a simple synchronous semantics but PSYNC programs run on asynchronous networks using a runtime system whose executions are indistinguishable from those of PSYNC.
- We prove that for any program $\mathcal{P}$, the runtime of $\mathcal{P}$ observationally refines $\mathcal{P}$, assuming clients with commutative operations.
- We prove that, for an important class of specifications including consensus, if a PSYNC program satisfies the specification, then its runtime system satisfies it as well.
- We have implemented and verified several fault-tolerant algorithms using PSYNC. We evaluate the *LastVoting* [24] that corresponds to Paxos [47] in the HO-model, in a distributed key-value store, and show that the PSYNC implementation performs comparably to high-performance consensus implementations.

```
interface
    init(v: Int); out(v: Int)

variable
    x: Int; ts: Int; vote: Int
    ready: Boolean; commit: Boolean
    decided: Boolean; decision: Int

//auxiliary function: rotating coordinator
def coord(phi: Int): ProcessID =
    new ProcessID((phi/phase.length) % n)

//initialization
def init(v: Int) =
    x := v
    ts := -1
    ready := false
    commit := false
    decided := false
```

```
1  val phase = Array[Round]( //the rounds
2    Round /* Collect */ {
3      def send(): Map[ProcessID, (Int,Int)] =
4          return MapOf(coord(r) → (x, ts))
5      def update(mbox: Map[ProcessID, (Int,Int)]) =
6          if (id = coord(r) ∧ mbox.size > n/2)
7              vote := mbox.valWithMaxTS
8              commit := true },
9    Round /* Candidate */ {
10     def send(): Map[ProcessID, Int] =
11         if (id = coord(r) ∧ commit) return broadcast(vote)
12         else return ∅
13     def update(mbox: Map[ProcessID, Int]) =
14         if (mbox contains coord(r))
15             x := mbox(coord(r))
16             ts := r/4 },
17   Round /* Quorum */ {
18     def send(): Map[ProcessID, Int] =
19         if ( ts = r/4 ) return MapOf(coord(r) → x)
20         else return ∅
21     def update(mbox: Map[ProcessID, Int]) =
22         if (id = coord(r) ∧ mbox.size > n/2)
23             ready := true },
24   Round /* Accept */ {
25     def send(): Map[ProcessID, Int] =
26         if (id = coord(r) ∧ ready) return broadcast(vote)
27         else return ∅
28     def update(mbox: Map[ProcessID, Int]) =
29         if (mbox contains coord(r) ∧ ¬decided)
30             decision := mbox(coord(r))
31             out(decision)
32             decided := true
33         ready := false
34         commit := false })
```

A simplified version of *LastVoting* in PSYNC. The program has four rounds, which execute in a loop. r contains the round number. The function `coord(r)` returns the identity of the coordinator of round r. Its identity changes between phases. In one phase, the coordinator collects proposals from the other replicas, picks one of them (`Collect`), and tries to impose it to the other replicas (`Candidate`). If a majority of processes agree with the coordinator's proposal (`Quorum`) then eventually all processes will accept this value as their decision (`Accept`).

Figure 1: The *LastVoting* consensus algorithm in PSYNC

Sec. 2 introduces PSYNC using an example. Sec. 3 defines indistinguishability and observational refinement. Sec. 4 defines the domain specific language PSYNC, while Sec. 5 describes its runtime system. The verification techniques are presented in Sec. 6. Finally, Sec. 7 presents the experimental evaluation of PSYNC.

2. Overview

In this section we present the main features of PSYNC using a consensus service. The service uses the *LastVoting* [24] algorithm, shown in Fig. 1, an adaptation of the Paxos algorithm to the HO-model.

We want to build a distributed read-write register. The application needs to give the users a coherent view of the data. Read requests can be served locally, but write requests need to be agreed upon globally. To agree on the order of requests, or sets of requests (batching), the application uses a consensus service.

The consensus service exports event-driven operations that the clients use to communicate with the service. These operations define the interface of the service. The interface of a service implemented in PSYNC has two main types of operations: input operations, denoted `init`, that the clients use to send a request to the service, and output operations, denoted `out`, that the service uses to reply to a client request.

For example, a client sends a request to the *LastVoting* program using `init`(v), where v is the new value the client wants to write to the register. The service replies to the client by generating an output event `out`(v') which contains the new value of the register.

A service process that receive a client request, starts executing *LastVoting* by calling its initialization function `init`. We assume that the clients are constantly sending new requests so each process has an initial value. Each replica stores the value it believes should be written in the variable `x`. After the initialization phase replicas typically have different `x` values. The goal of the algorithm is to make the replicas agree on one of these values. The execution of *LastVoting* terminates when all replicas agree.

LastVoting roughly works by first establishing a majority of processes that agree on the same value. A designated process, called the coordinator, collects proposals for the value of `x` from the other replicas and picks one of them (execution of the round `Collect`, Line 2). In the next round, `Candidate` (Line 9), the coordinator tries to impose the chosen value to a majority of replicas. The round `Quorum` (Line 17) checks that this majority has been correctly established. If a quorum is formed then a decision is made, and all replicas will accept it as the `decision` value, during the `Accept` round (Line 24). The execution of these rounds repeats, starting again with the `Collect` round, possibly with a different coordinator. The crux for the correctness of the algorithm is that strict majorities have a non-empty intersection. Since a decision requires the agreement of a majority of processes, the decision is unique.

Program structure. A PSYNC program is composed of an interface, a set of local variables, an initialization function, and a sequence of rounds. Each round defines a `send` and an `update` method. All processes execute in lockstep the same round. A part from the local variables processes use build-in read-only variables: `r` represents the round number, `n` the number of processes in the system, and `id` is the process unique identifier.

The `init` method takes as argument an integer. The `send` function returns the set of messages sent by the process executing it during the current round. A message is a pair (payload, recipient). The payload type might differ across rounds. For example, in `Collect` processes send pairs of integers while in `Quorum` they send one integer. The recipient is identified by a `ProcessId`. PSYNC supports broadcast and point-to-point communication. The statement `broadcast(T)` returns a set of messages of type (`T`, `ProcessId`), where `T` is the type of payload. This set contains one message for every process in the system (all messages have the same payload).

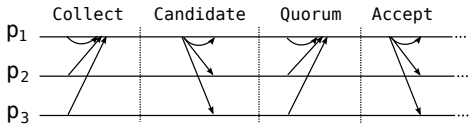


Figure 2: Execution of the *LastVoting* in a synchronous environment without faults. Process $\mathtt{p}_1$ is the coordinator. Dotted lines represent the boundaries between rounds.

The `update` takes as argument the set of messages received during the current round and modifies the local state of the process. The input of `update` is a set of pairs of type (`T`, `ProcessId`), where `T` is the type of the payload and the second component is the sender's identity. The `update` method can use external functions to perform sequential computations and modify the local state. For example, `mbox.valWithMaxTs` used in `Collect` scans the set of received messages and returns a value v such that (v, t) belongs to `mbox` and for any other (v', t') in `mbox`, $t' \leq t$. The functions `_1` and `_2` project a pair on its first resp. second component.

Execution. A run of a PSYNC program starts with a call to the initialization function `init` (on each process). In *LastVoting* the `init` function initializes the value of `x` known locally by each replica. The run continues with processes repeatedly executing, in lock step, the sequence of rounds defined by `phase` in the program.

In an ideal system (where no messages are lost or delayed) an execution of one phase of the *LastVoting* program would result in the trace shown in Fig. 2. The processes proceed in lockstep, messages are delivered in time, and agreement is reached after four rounds. In reality an execution is more likely to look like the one shown in Fig. 3a, due to different delivery times for messages, different processors speeds, and crashes.

To reason about asynchrony and faults, the PSYNC semantics is based on the HO-model. Each process has a variable interpreted over sets of processes, called HO-set. The set of messages received by a process p in round r, is the set of messages that were sent to p by the processes in its HO-set. At the beginning of each round, HO-sets are non-deterministically modified by the environment.

The HO-model captures asynchronous behaviors and faults while providing the illusion of a lockstep semantics. Therefore, it is possible to reflect the faults in Fig. 3a using a lockstep semantics by setting the HO sets to the appropriate values. Fig. 3b shows a lockstep execution where each process receives the same set of messages as in Fig. 3a. If a message is dropped by the network, e.g., the message p_2 sends to p_1 in `Collect`, then p_2 is not included in the HO of the p_1. If a message is delayed far too long, then the sender is not included in the HO-set of the receiver. For example, in `Quorum`, the coordinator decides on a value if a majority of processes agrees with its proposal. Therefore, the coordinator moves to the next round, without waiting for acknowledgements from all processes. In Fig. 3a the coordinator p_1 starts executing round `Accept` despite not receiving the acknowledgement sent by $\mathtt{p}_3$. However, this acknowledgement is eventually delivered when the coordinator is in the fourth round, but the message comes too late to influence the local computation. Therefore, it is as if the message was lost. In the lockstep execution, Fig. 3b, p_3 is not included in the HO-set of p_1 in the `Quorum` round. Finally, crashes do not directly impact the view of correct processes. Crashed processes are modeled using correct process which are not included in the HO-set of any other process. It is as if all messages they send are dropped after the instant of the actual crash.

Specification and Verification. We have developed a state-based verification engine for PSYNC programs. The specification of *LastVoting* includes properties like agreement, which say that all processes decided on the same value:

$$\Box(\forall p, p'.\ p.\mathtt{decided} \wedge p'.\mathtt{decided} \Rightarrow p.\mathtt{decision} = p'.\mathtt{decision}),$$

where p, p' are processes and $p.\mathtt{decided}$, $p.\mathtt{decision}$ is the value of the local variable `decided`, resp. `decision`, of process p.

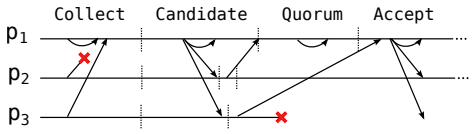
The verification engine is based on deductive verification. This assumes that the program is annotated with inductive invariants and the engine proves the validity of the annotations and the fact that they imply the specification. The lockstep semantics is essential in order to have simple inductive invariants. For instance, the crux of the invariant that shows agreement is the existence of a majority of processes that agree on a value v when at least one process decides:

$$\forall p.\ p.\mathtt{decided} = \mathit{false} \vee$$
$$\exists v, t, A.\ A = \{p \mid\ p.\mathtt{ts} \geq t\} \wedge |A| > n/2 \wedge \forall p.\ p \in A \Rightarrow p.\mathtt{x} = v.$$

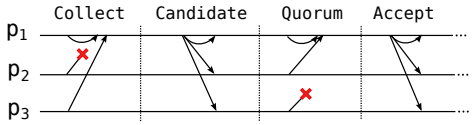
Beyond safety properties, we are also interested in proving liveness properties, such as, $\Diamond(\forall p.\ p.\mathtt{decided})$. Because consensus is not solvable in asynchronous networks with faults, the network must satisfy liveness assumptions to ensure progress of the algorithm. The liveness assumptions impose constraints, typically lower bounds on the cardinality of the HO-set. For example, proving that *LastVoting* eventually makes a decision requires a sequence of four rounds starting with `Collect` during which the environment picks values for the HO-set such that:

$$|\mathsf{HO}(\mathtt{coord}(r))| > n/2 \wedge \forall q.\ \mathtt{coord}(r) \in \mathsf{HO}(q). \qquad (1)$$

For fault-tolerant distributed algorithms, the specification, the liveness assumptions, and the inductive invariants, require reasoning about set comprehensions, cardinality constraints, and process quantification. To express these properties and check their validity we use a fragment of first-order logic, called $\mathbb{CL}$, and its semi-decision procedure [30]. The complete *LastVoting* program, its specification and annotations are available in Appendix **??**.



(a) An asynchronous, faulty execution of the *LastVoting*



(b) Corresponding indistinguishable lockstep execution

Figure 3: Correspondence between the semantics and execution

3. Indistinguishability and Observational Refinement

In this section we formally define in terms of transition systems the main theoretical concepts used in the paper: indistinguishability, observational refinement, and the relationship between them.

3.1 Definitions

A *transition systems* is a tuple $TS = (P, V, A, s_0, T)$, where P is a set of processes, V is a finite set of variables, $V = \bigcup_{p \in P} V_p$ with V_p the set of local variables of process $p \in P$, A is a (possibly infinite) set of labels, $A = \bigcup_{p \in P} A_p$ with A_p the set of transition labels of process $p \in P$, $s_0 \in \Sigma$ is the initial state of the system, T is a transition relation $T \subseteq \Sigma \times 2^A \times \Sigma$ where the label of each

transition contains at most one label per process. The state space of TS is $\Sigma = [P \to V \to \mathcal{D}] \uplus \{*\}$ where $*$ is a special state disjoint from the other ones and $\mathcal{D}$ the data domain where variables are evaluated. A state $s \in \Sigma$ is a valuation of the processes variables. Given a process $p \in P$, $s(p)$ is the local state of p, which is an evaluation of p's local variables, i.e., $s(p) \in [V_p \to \mathcal{D}]$.

An *execution* of a system TS is an infinite sequence $s_0 A_0 s_1 A_1 \ldots$ such that for all $i \geq 0$, $s_i \in \Sigma$, $A_i \subseteq A$ ($A_i \neq \emptyset$) such that each process takes at most one transition, and the state s_{i+1} is a successor of the state s_i in the transition relation T, i.e., $(s_i, A_i, s_{i+i}) \in T$. We denoted by $[\![TS]\!]$ the set of executions of the systems TS. A *run* is the sequence of states induced by an execution and a *trace* the sequence of labels induced by an execution. We denoted by $\mathsf{Runs}(TS)$, $\mathsf{Traces}(TS)$, the set of runs, respectively the set of traces, of a transition system TS.

The *projection* of execution π on a process p, denoted $\pi \downarrow_p$, is obtained from π by keeping only the state of the variables and transition labels local to p, i.e., $\pi\downarrow_p = s'_0 A'_1 s'_1 \ldots$ where $s'_i = s_i(p)$ and $A'_i = A_i \cap A_p$.

Let π be an alternating sequence over the alphabet Σ and A. A block over Σ is a word $s(\emptyset s)^*$ where $s \in \Sigma$ and $\emptyset$ is a transition with an empty label. The *stuttering* closure of π is the set of executions in the language generated by replacing every state in π with the corresponding block. We write $\pi_1 \equiv \pi_2$ iff the execution π_1 is equivalent to the execution π_2 up to stuttering, i.e., there is an execution in the stuttering closure of both π_1 and π_2.

3.2 Indistinguishability

We define indistinguishability, an equivalence relation between executions of a transition system. Furthermore, we lift this definition to relate executions of different transition systems.

Definition 1 (Indistinguishability)**.** *Given two executions π and π' of a system TS, a process p cannot distinguish locally between π and π', denoted $\pi \simeq_p \pi'$ iff the projection of both executions on the local states and transitions of p agree up to finite stuttering, i.e., $\pi\downarrow_p \equiv \pi'\downarrow_p$.*

Two executions π and π' are indistinguishable*, denoted $\pi \simeq \pi'$, iff no process can distinguish between them, i.e., $\forall p \in P.\ \pi \simeq_p \pi'$.*

To relate executions of different systems indistinguishability is extended, as expected, to $\simeq_{W,L}$ where W, L, are subsets of the common variables and labels of the two systems.

Definition 2 (Indistinguishable systems)**.** *A system TS_1 is indistinguishable from a system TS_2 denoted $TS_1 \trianglerighteq TS_2$ iff they are defined over the same set of processes and for any execution $\pi \in [\![TS_1]\!]$ there exists an execution $\pi' \in [\![TS_2]\!]$ such that $\pi \simeq_{W,L} \pi'$ where $W = V_1 \cap V_2$ and $L = A_1 \cap A_2$.*

Notice that indistinguishability takes into account only the local process order of transitions. For example, the two executions in Fig 3 are indistinguishable, although the `Quorum` round is executed during non-overlapping periods of time by process p_1 and p_2 in Fig. 3a while in Fig. 3b the order between them is lost.

3.3 Observational refinement

A client and a service synchronize through an *interface* given by the shared transition labels. When a client uses a PSYNC program as a service, e.g., the *LastVoting* as a consensus service, the client should be able to use indistinguishable systems interchangeably.

A distributed *client* of a service is a transition system defined by the parallel compositions of a set of n transitions systems over disjoint alphabets, each of this transition systems representing one client process in the system. This definition implies that transitions of different clients processes commute, which intuitively means client processes do not synchronize with each other directly. The full definition is in Appendix B.

Composition between a client and a service. We assume the network is formed of different nodes, such that each node consists of a client process and a service process. A client sends requests to the service process located on the same node. For presentation reasons we assume that the transitions system associated with the client respectively with the service use the same set of processes, corresponding to the nodes in the network.

Let $\mathcal{S} = (P, V', A' \uplus I, s'_0, T')$ be a service of interface I. A client $C = (P, V, A, s_0, T)$ connects to the service $\mathcal{S}$ iff $I \subseteq A$ and $A \cap A' = \emptyset$. We consider $I = \uplus_{p \in P} I_p$ where I_p is the labels of transitions only taken by the process p, in both $\mathcal{S}$ and C.

We define the behaviors of client C that interacts with service $\mathcal{S}$ as the set of executions of a transition system denoted $C(\mathcal{S})$. The system $C(\mathcal{S})$ is obtained by (1) taking the asynchronous product between C and $\mathcal{S}$ and forcing that any transitions labeled by $b \in I$ is always taken simultaneously by C and $\mathcal{S}$, and (2) projecting out $\mathcal{S}$ from the product (we are only interested in the client), (3) After projecting out $\mathcal{S}$, we remove the transitions with no labels. We obtain a transition system with the same state space as the original client, but with fewer behaviors.

For example, a client communicates with *LastVoting* in Fig. 1 using the interface $I_{\texttt{init,out}} = \{\texttt{init}_p(v), \texttt{out}_p(v) \mid p \in P, v \in \mathbb{Z}\}$. For the client, an $\texttt{init}_p(v)$ transition sends a request to process p executing *LastVoting* and makes the client wait for the reply $\texttt{out}_p(v')$. An $\texttt{init}_p(v)$ transitions of *LastVoting* corresponds a call to the initialization function `init` on process p with v as input parameter. Similarly $\texttt{out}_p(v')$ transitions are used to handle the replies between the service process p and the client process p.

In general there are different possible implementations of a service. In the following we define a refinement relation between service implementations.

Definition 3 (Observational Refinement)**.** *Let TS_1 and TS_2 be two transition systems and a common interface I. Then, TS_1* refines *TS_2 w.r.t. I denoted $TS_1 \sqsubseteq_I TS_2$, if for any client C,*

$$\mathsf{Runs}(C(TS_1)) \subseteq \mathsf{Runs}(C(TS_2)).$$

We say that TS_1 observationally refines TS_2 if every run of a client that uses TS_1 is also a run the same client using TS_2.

Moreover, since we consider clients which do not impose an order between the transitions on different client processes, indistinguishability is equivalent with observational refinement.

Theorem 1. *Let TS_1 and TS_2 be two systems with a common interface I. If $TS_1 \trianglerighteq TS_2$ then $TS_1 \sqsubseteq_I TS_2$.*

Proof. (Sketch) The proof is based on the Corollary 43 from [34], which states that sequentially consistency is equivalent with observational refinement when client transitions commute across processes. The main ingredient to extend this corollary is that indistinguishability is equivalent with sequential consistency when one of the systems is sequential. More precisely, if TS_2 is a sequential transition system, then $TS_1 \trianglerighteq TS_2$ iff TS_1 is sequentially consistent with TS_2. □

4. Syntax and semantics of PSYNC

PSYNC is designed as a domain specific language embedded within a general purpose programming language. PSYNC offers specialized syntactic constructs for distributed algorithms. In this section we define only the core communication and synchronization control structures using an abstract syntax and we focus on the types of the operations composing a PSYNC program.

Syntax. We give an abstract syntax for PSYNC programs in Fig. 4. A program has an interface, a number of local variables, an initial-

$$
\begin{array}{rcl}
\textit{program} & ::= & \textit{interface variable}^* \textit{ init phase} \\
\textit{interface} & ::= & \texttt{init}\colon \textit{type} \to () \quad (\textit{name}\colon \textit{type} \to ())^* \\
\textit{variable} & ::= & \textit{name}\colon \textit{type} \\
\textit{init} & ::= & \texttt{init}\colon \textit{type} \to () \\
\textit{phase} & ::= & \textit{round}^+ \\
\textit{round}_{\mathsf{T}} & ::= & \texttt{send}\colon () \to [P \mapsto \mathsf{T}] \quad \texttt{update}\colon [P \mapsto \mathsf{T}] \to ()
\end{array}
$$

Figure 4: PSYNC abstract syntax.

ization operation `init`, and a non-empty sequence of rounds, called phase. Each process executes in a loop the sequence of rounds defined in the phase.

The interface is a set of functions, which are the transitions observed by the client. The interface includes an `init` event that corresponds to calling the `init` operation. A client request is received using an `init` labeled transition. The other functions are outputs of the program to the client. We denote by $name_{\mathrm{p}}$, a label in the interface when the associated transition is executed by process p.

Each round is parameterized by the type T which represents the payload of the messages. The `send` and `update` operations of a round must agree on T. However, different rounds can have different payload types. For instance, the payload of `Collect` in Fig 1 is `(Int,Int)` and `Int` for the other type of rounds.

The operations in a PSYNC program do not use directly any iterative control structures. For complex sequential computations they can use auxiliary operations implemented in the host language. The `init`, `send`, and `update` operations are assumed to terminates within a number of steps that depends on the number of processes and the input values of the initialization function. PSYNC is designed to facilitate the implementation of message passing concurrency. Proving total correctness of the sequential code executed by each process is orthogonal to the scope of PSYNC.

Semantics. Assuming a finite, non-empty set of n processes P, the state of a PSYNC program is represented by the tuple $\langle SU, s, r, msg, \mathsf{HO}\rangle$ where:

- $SU \in \{Snd, Updt\}$ indicates whether the next operation is send or update;
- $s \in [P \to V \to \mathcal{D}]$ stores the local states of the processes;
- $r \in \mathbb{N}$ is a counter for the number of executed round;
- $msg \subseteq 2^{P,\mathsf{T},P}$ stores the messages in the system between the SEND and UPDATE phase of a round;
- $\mathsf{HO} \in [P \to 2^P]$ evaluates the HO-sets for the current round.

The semantics of a PSYNC program is shown in Figure 5.

A transition is written as $S \xrightarrow{I,O} S'$ where S, S' are states, O is a set of labels from the interface, corresponding to observable transitions, I is a set of labels not in the interface corresponding to internal transitions. A client of a PSYNC program can only observe the transition labels in O.

We consider the following shorthands: $|\texttt{phase}|$ is the number of rounds in a phase and $\texttt{phase}[r]$ is used to identify the $(r \mod |\texttt{phase}|)$ round in a phase. For example, Fig. 1 declares $|\texttt{phase}| = 4$ types of rounds and $\texttt{phase}[3]$ identifies a `Quorum` round. The operation m of a round $\texttt{phase}(r)$ is $\texttt{phase}(r).m$. A transitions $s(p) \xrightarrow{op,o} s'(p)$ says that p executes operation op in local state $s(p)$ and reaches local state $s'(p)$. The execution of op produces an observable transition o, i.e., o is in the interface.

Initially the state of the system is undefined, denoted by $*$, and the first transition of every process p is to cal the `init` operation whose arguments are received from the client via an `init` transition executed by the process p (see INIT in Fig. 5). The `init` operation does not return a value but initialize the state of the system. Initially, the round counter is 0, there are no messages in the system, and the

INIT

$$
\frac{\forall p \in P.\ * \xrightarrow{\texttt{init}(v_p)} s(p)}{* \xrightarrow{\emptyset,\{\texttt{init}_p(v_p)\mid p\in P\}} \langle Snd, s, 0, \emptyset, \mathsf{HO}\rangle}
$$

SEND

$$
\frac{\begin{array}{c}\forall p \in P.\ s(p) \xrightarrow{\texttt{phase}[r].\texttt{send}(m_p)} s(p) \\ msg = \{(p,t,q) \mid p \in P \wedge (t,q) \in m_p\}\end{array}}{\langle Snd, s, r, \emptyset, \mathsf{HO}\rangle \xrightarrow{\{\texttt{send}_p(m_p)\mid p\in P\},\emptyset} \langle Updt, s, r, msg, \mathsf{HO}'\rangle}
$$

UPDATE

$$
\frac{\begin{array}{c}\forall p \in P.\ mbox_p = \{(q,t) \mid (q,t,p) \in msg \wedge q \in \mathsf{HO}(p)\} \\ \forall p \in P.\ s(p) \xrightarrow{\texttt{phase}[r].\texttt{update}(mbox_p),o_p} s'(p) \\ r' = r+1 \qquad O = \{o_p \mid p \in p\}\end{array}}{\langle Updt, s, r, msg, \mathsf{HO}\rangle \xrightarrow{\{\texttt{update}_p(mbox_p)\mid p\in p\},O} \langle Snd, s', r', \emptyset, \mathsf{HO}\rangle}
$$

Figure 5: PSYNC semantics.

first operation is Snd. An execution alternates SEND and UPDATE transitions from Fig 5.

During a SEND transition, the messages sent by each process are added to the message buffer msg. The messages in msg are triples of the form (sender, payload, recipient), where the sender and receiver are processes and the payload has type `T`. The `send` operation does not affect the state of the processes. The values of the HO-sets are defined non-deterministically by the environment.

In an UPDATE step, messages are received and the `update` operation is applied locally on each process. The set of received messages is the input of `update`. A message is received only if the sender is in the receiver's HO-set. The `update` operation might produce an observable transition o_p. At the end of the round msg is purged and r is incremented by 1.

To obtain a transition system as in Section 3, the fields SU, r, HO are copied locally on each process, the interface along the `send` and `update` operations define the labels of the transition system and the pool of messages msg is represented by a special *network* process whose only local variable is msg.

Environment assumptions. Many problems, such as consensus, are not solvable in asynchronous networks with faults. More precisely, the existence of an algorithm solving such a problem, is predicated by assumptions on the network on top of which the algorithm is executed. Therefore, the algorithm designer must provide not only a PSYNC program but also the assumptions on the network its algorithm is designed for.

In PSYNC the network assumptions translate into assumptions on the environment actions. They are given as linear temporal logic (LTL) formulas over atomic propositions that constrain the values of the HO-sets. The classic taxonomy of distributed systems distinguishes the synchrony degree of the network, the reliability of the links, different types of process failure. The relation between the classic types of systems and the corresponding assumptions on the HO-sets is given in []. For example, if an algorithm is designed for asynchronous network with reliable links and at most f crash failures then in PSYNC the algorithm designer should assume an environment that assigns sets of cardinality greater than or equal to $n - f$ to each HO-set, i.e., $\forall p.|\mathsf{HO}(p)| \geq n - f$, where n is the number of processes in the network.

We consider that network assumptions are required only to guarantee the desired liveness properties. Without making any assumption on the environment a program might never make any progress. For example, the environment can decide that the HO-sets are always empty and no message is delivered. In order to ensure termi-

nation *LastVoting* assumes that eventually there exists a sequence of four rounds, starting with `Collect`, where (1) the coordinator is in the HO-set of any process, and (2) during the `Collect` and `Quorum` rounds of this sequence the HO-set of each process contains at least $n/2$ processes. Formally, this environment assumption is expressed by the formula $\Diamond(\psi \wedge \circ(\psi \ \wedge \circ(\psi \wedge \circ\psi)))$, where ψ is given in 1. Notice that at the end of these four rounds, all processes have decided the value proposed by the coordinator in the `Collect` round, even if no process has made a decision previously.

Definition 4 (Execution)**.** *Given a* PSYNC *program $\mathcal{P}$ and a non-empty set of processes P, an execution of $\mathcal{P}$ is the sequence $*A_0 s_1 A_1 s_2 \ldots$ such that*

- $*A_0 s_1$ *is the result of the* INIT *rule;*
- $\forall i.\ s_i A_i s_{i+1}$ *satisfy the* SEND *or the* UPDATE *rule;*
- *the environment assumptions on* HO*-sets are satisfied.*

The set of executions of $\mathcal{P}$ is denoted by $[\![\mathcal{P}]\!]$.

A lock-step execution of *LastVoting* has a finite prefix when the environment assumptions are not met followed by a suffix that satisfies them. In general, lock-step executions of PSYNC programs are an alternation of bad and good rounds, where a round is good if the corresponding environment assumptions are met.

For any program $\mathcal{P}$, we consider that the environment assumptions of $\mathcal{P}$ are time-invariant, i.e., they are of the form $\Box\Diamond\varphi$, where φ is the network assumption that $\mathcal{P}$ relies on during its execution. Time invariance is important because we don't want the correctness of $\mathcal{P}$ to depend on the time it starts executing.

5. Runtime

In this section we define a runtime that executes PSYNC programs on a wide variety of network configurations. We start by describing the network assumptions, then the algorithm behind the runtime, and finally, we show that a client cannot distinguish between a PSYNC program and its runtime execution.

5.1 Partial synchrony

The round structure can be accurately simulated on an asynchronous network provided that the latter satisfies the partial synchrony assumption. Partial synchrony means that the network alternates between bad and good time periods, where during a good period the communication and the processes are synchronous while through a bad period they are asynchronous. The good periods are needed to ensure progress of the system and the safety properties should hold during arbitrarily long asynchronous periods.

Definition 5 (Synchronous network)**.** *A network is synchronous if there exists Θ and Δ two positive integers, such that:*

- *Θ is the minimal time interval in which any process is guaranteed to take a step;*
- *Δ is the maximal transmission delay between any processes.*

Intuitively, Θ corresponds to process synchrony and Δ is communication synchrony. Both are required in order to solve problems like consensus [**?**]. When the network is asynchronous there are no bounds on the communication delay and relative speed.

Remark 1. *The transmission delay is typically much larger than the time required to ensure a computation step, i.e., $\Delta \gg \Theta > 0$.*

Definition 6 (Partially synchronous network [31][1])**.** *A network is* partially synchronous *if the constants Θ, Δ exist, are known, and eventually hold after a time gst, called the global stabilization time.*

[1] In [31] a second definition is proposed where Θ and Δ, are unknown but they hold from the beginning. Our results hold under both definitions of partial synchrony. We have chosen Def. 6 for performance reason.

Def. 6 characterizes the partial synchrony assumption w.r.t. an initial time. We assume this time coincide with the start of the program execution. To avoid depending on the time processes start, a network is partially synchronous if it alternates between good and bad time periods, where the good periods are as long as needed. The required length of good periods depends on the program. Def. 6 takes the supremum of good periods for any program to allow a general statement about liveness of the system.

5.2 Runtime Algorithm

The runtime system is defined by the asynchronous composition of all processes in the network intersected with the partial synchrony assumptions. Given a PSYNC program $\mathcal{P}$, Fig. 6 shows the code executed by each process to run $\mathcal{P}$. Roughly, processes execute locally the same sequence of rounds as in $\mathcal{P}$, but the parallel composition is asynchronous.

The algorithm in Fig. 6 uses two while-loops. One iteration of the outer loop (line 9) executes one round. The inner loop (line 15) accumulates messages until a timeout is reached. Because the network is not synchronous, the runtime deals with messages of past or future rounds, i.e., messages tagged with round numbers strictly smaller or bigger than the process's current round number. Late messages are dropped (line 18), and a message from a future round forces the runtime to execute the outer loop until it catches up and reaches the round of the received message (line 11). The send and update operations, on line 10 and 26, are those defined in the executed PSYNC program $\mathcal{P}$. To deal with messages duplication the accumulated messages are stored in a set. The function `tryReceive(d)` tries to receive a message if one is available, or becomes available during the next `d` time units. If no message is available in this period then the method returns $\perp$.

The variable `to` has the same reference value across processes. It is used to measure locally a time interval of length `to` in reference time units. The function `currentTime` returns the value of a local clock, s.t., all processes measures the same duration for a time interval of length `to`. We assume that the processor's speed is not related to its clock and also we assume a bounded clock drift across processes. Therefore, processes can execute a different number of instructions while measuring the same interval.

Messages are tuples of the form $(sender, payload, receiver, round)$, where $sender, receiver$ are the respective sender and receiver of the message, $payload$ is the content of the message, and $round$ is the sender's round number when the message was sent.

In the following, given a PSYNC program $\mathcal{P}$, we define the semantics of the runtime of $\mathcal{P}$. A state of the $Runtime$ of $\mathcal{P}$ is represented by the tuple $\langle s \uplus s_r, msg\rangle$ where:

- $s \in [P \to V \to \mathcal{D}]$ is an evaluation of the variables in $\mathcal{P}$;
- $s_r \in [P \to V_r \to \mathcal{D}]$ is an evaluation of the variables introduced by the runtime;
- $msg \in [(P, \mathsf{T}, P, \mathbb{N}) \to \mathbb{N}]$ is a multiset of messages in transit.

In Fig. 7 we define the semantics of the most important instructions of the algorithm in Fig. 6. Each transition has the form $s \xrightarrow{I,O} s'$, where s, s are global states and I, resp. O, are the internal, resp. observable labels of the transition. The runtime of a program $\mathcal{P}$ interacts with a client using the interface of $\mathcal{P}$ defining the observable transitions. By considering msg a specific *network* process, we obtain a transition system as defined in Section 3.

Local process transitions. The SEND rule states that the messages sent by one process are tagged with the current round of the sender, i.e., $s(p).r$, and added to the global pool of messages, msg. The messages sent by one process in round r are defined by the `send` operation of the same round from $\mathcal{P}$. The RECEIVE1 and RECEIVE2 rules define the reception of a message. The RECEIVE2

```
1  //local variables
2  p                 //initialized PSync process
3  to                //timeout
4  r := 0            //current round number
5  msg := ⊥          //last received message
6  mbox := ∅         //messages received but not yet processed
7  t := currentTime() //time at which the current round began
8
9  while (true) {
10   p.phase[r % p.phase.size].send() //send event
11   if (msg ≠ ⊥ ∧ msg.round = r) {
12     mbox := {msg}
13     msg := ⊥
14   }
15   while (msg = ⊥ ∧ currentTime() < t + to) {
16     msg := tryReceive(t + to - currentTime()) //receive event
17     if (msg ≠ ⊥) {
18       if (msg.round < r) {
19         msg := ⊥
20       } else if (msg.round = r) {
21         mbox := mbox ∪ {msg}
22         msg := ⊥
23       }
24     }
25   }
26   p.phase[r % p.phase.size].update(mbox) //update event
27   r := r + 1
28   t := currentTime()
29   mbox := ∅
30 }
```

Figure 6: Algorithm to implement the round structure

rule describes a failed reception due to a timeout. The WAIT rule models a process waiting for a message. The UPDATE rule states that the semantics of `update` when called by a process whose current round is r is the the semantics of the `update` operation of round r from $\mathcal{P}$.

The CRASH, CRASHED, DUPLICATE, DROP describe the fault model. Progresses can crash, but do not recover. Messages can be duplicated and dropped by the network. The CRASHED rule states that crashed process do not modify the global state.

The system of transitions defined in Fig. 7 generates purely asynchronous executions and some of them are not possible under the partial synchrony assumptions. In order to integrate the partial synchrony of the network we need to reason about message delays and speed of the processes. We add a reference time to our system using a function τ that maps each state s_i in a trace π to a time in $\mathbb{N}$. τ is monotonic: $\forall i, j.\ i \leq j \Rightarrow \tau(s_i) \leq \tau(s_j)$. There cannot be infinite subsequence of π where the time increase is finite.

An execution $\pi = s_0 A_0 s_1 A_2 \dots$ of the transition system in Fig. 7 satisfies the partially synchrony assumption on the network iff there exits a global stabilization time, gst, s.t. $\pi = \pi_a \pi_s$ and

- for any $s \in \pi_s$, $\tau(s) > gst$;
- for any i, j such that $\tau(s_i) > gts$ and $\tau(s_j) - \tau(s_i) \geq \Theta$, every process takes at least one step the sequence $s_i \dots s_j$;
- for any i,j such that $\tau(s_i) > gst$, $\mathtt{send}_p(ms) \in A_i$, and $receive_q(m) \in A_j$ with $m \in ms$, either the delay is $0 < \tau(s_j) - \tau(s_{i+1}) \leq \Delta$ or let $k \in [i, j]$ the first state with $\tau(s_k) - \tau(s_{i+1}) > \Delta$ then q does not take any WAIT or RECEIVE2 step in the $s_k \dots s_j$ interval;
- there is no message duplication after gts[2].

[2] Limiting duplication only simplifies the proof. It is possible to tolerate a bounded amount of duplication after gts. The time to process the duplicate messages has to be factored in the timeout.

Local transitions

SEND

$$\frac{p \xrightarrow{\mathtt{send}(ms)} p' \qquad msg' = \{(p, m, q, p.r) \mid (m, q) \in ms\} \cup msg}{\langle p, msg\rangle \xrightarrow{\{\mathtt{send}_p(ms)\},\emptyset} \langle p', msg'\rangle}$$

RECEIVE1

$$\frac{m \in msg \qquad m.receiver = p \qquad p \xrightarrow{receive(m)} p' \qquad msg' = msg \setminus m}{\langle p, msg\rangle \xrightarrow{\{receive_p(m)\},\emptyset} \langle p', msg'\rangle}$$

RECEIVE2

$$\frac{\forall m \in msg.\ m.receiver \neq p \qquad p \xrightarrow{receive(\bot)} p'}{\langle p, msg\rangle \xrightarrow{\{receive_p(\bot)\},\emptyset} \langle p', msg\rangle}$$

UPDATE

$$\frac{m = p.mbox \qquad p \xrightarrow{\mathtt{update}(m),\alpha_p} p'}{\langle p, msg\rangle \xrightarrow{\{\mathtt{update}_p(m)\},\{\alpha_p\}} \langle p', msg\rangle}$$

DROP

$$\frac{ms' \subset ms}{\langle p, ms\rangle \xrightarrow{\emptyset,\emptyset} \langle p, ms'\rangle}$$

CRASH

$$\frac{s(p) \neq \bot \qquad s' = s[p \leftarrow \bot]}{\langle s, msg\rangle \xrightarrow{\emptyset,\emptyset} \langle s', msg\rangle}$$

WAIT

$$\frac{p \neq \bot}{\langle p, msg\rangle \xrightarrow{\{wait_p\},\emptyset} \langle p, msg\rangle}$$

CRASHED

$$\frac{p = \bot}{\langle p, msg\rangle \xrightarrow{\{\bot_p\},\{\bot_p\}} \langle p, msg\rangle}$$

DUPLICATE

$$\frac{m \in ms \qquad ms' = ms \cup \{m\}}{\langle p, ms\rangle \xrightarrow{\emptyset,\emptyset} \langle p, ms'\rangle}$$

Global transitions

INIT

$$\frac{\forall p \in P.\ * \xrightarrow{\mathtt{init}(v_p)} s(p)}{* \xrightarrow{\emptyset,\{\mathtt{init}_p(v_p) \mid p \in P\}} \langle s, \emptyset\rangle}$$

PARALLEL COMPOSITION

$$\frac{\begin{array}{c} P' \subseteq P \qquad msg = \uplus_{p \in P'} msg_p \qquad msg' = \cup_{p \in P'} msg'_p \\ I = \cup_{p \in P'} I'_p \qquad O = \cup_{p \in P'} O'_p \qquad \forall p \in P \setminus P'.\ s(p) = s'(p) \\ \forall p \in P'.\ \langle s(p), msg_p\rangle \xrightarrow{I_p, O_p} \langle s'(p), msg'_p\rangle \end{array}}{\langle s, msg\rangle \xrightarrow{I,O} \langle s', msg'\rangle}$$

Figure 7: Runtime semantics of a PSYNC program $\mathcal{P}$. The local transitions that involve a single process, the global ones combine them. We emphasise the most important local transition of a process executing the algorithm in Fig. 6. The rules SEND, RECEIVE, and UPDATE correspond to Line 10, 16, and 26 of the algorithm. The semantics of `send` and `update` is given by the program $\mathcal{P}$.

Definition 7 (Executions of the runtime system)**.** *We define the set of executions of the runtime system associated with a* PSYNC *program $\mathcal{P}$, denoted $[\![\mathcal{P}]\!]_{rt}$, by the set of partially synchronous executions of the transitions system in Fig. 7 which satisfy the environment assumptions defined in $\mathcal{P}$.*

Remark 2. *The environment assumptions are time invariant, i.e., they are LTL formulas of the form $\Box\Diamond\varphi$. Therefore we consider $\pi = \pi_a \pi_s \in [\![\mathcal{P}]\!]_{rt}$ satisfies the environment assumption iff π_s satisfies $\Box\Diamond\varphi$.*

The runtime does not include HO-sets. To evaluate the environment assumptions from $\mathcal{P}$ on the runtime executions, HO-sets are replaced with the set of received messages. That is, any constraint on $\mathsf{HO}(p)$ in round r, is replaced by the same constraint over

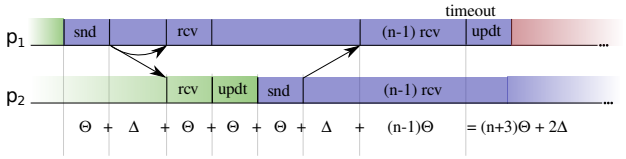


Figure 8: Processes synchronizing. The colors represent rounds.

$\texttt{mbox}_p$, the set of messages received by process p in round r. The `update` called by process p in round r takes $\texttt{mbox}_p$ as argument.

In the following we identify values for the timeout variable which ensure that any execution of the runtime implements correctly the round structure under the partially synchrony assumption allowing the reception of all the messages that can be received in a round. We begin with the value of the timeout when the network is synchronous and then we compute the timeout for partially synchronous networks. We assume that every process periodically communicates with every other process. If the algorithm does not send messages, the runtime inserts heatbeat messages. Communication is needed for synchronization.

Definition 8 (Synchronous execution)**.** *An execution of $[\![\mathcal{P}]\!]_{rt}$ is called* synchronous *if in any time interval of length Θ each process takes at least a* `send`*,* `update`*, or receive step and every message sent in a given round is received in the same round, dropped by the network, or the receiver has crashed.*

Lemma 1. *If the network is synchronous any execution of $[\![\mathcal{P}]\!]_{rt}$ is synchronous if the value of the timeout is $(n+1)\Theta + \Delta$.*

When the network is synchronous the maximal duration of a round is given by the slowest process and it equals $(n+1)\Theta + \Delta$ reference time units. Since the timeout equals $(n+1)\Theta + \Delta$ it slows down the fasters process such that the slowest process will have enough time to process all the messages sent in current round before the fastest process moves to the next round.

Without appropriate timeout values an execution of the runtime can diverge without processes ever synchronizing, due to the asynchrony of the network in the beginning of the execution.

Lemma 2. *If `to` $> (n+3)\Theta + 2\Delta$ and the network is partially synchronous, any execution π of $[\![\mathcal{P}]\!]_{rt}$ is of the form $\pi = \pi_a\pi_s$ where π_a is finite and π_s is synchronous.*

Proof. Let s be the first state such that $\tau(s) \geq gst$. We divide the asynchronous prefix in two parts $\pi_a = \pi_1\pi_2$ with π_1 is a finite prefix and π_2 the part of the execution starting in a state s. We show that the length of π_2 is bounded.

The actual computation time for a round is $(n+2)\Theta$, the remaining time $\Theta + 2\Delta$ is the slack required to take into account message delays and relative speed. After gts, the processes can use the slack time in each round to process more messages than the number of messages sent during that round. Because π_1 is finite, the number of messages that separate the fastest process from the slowest is also finite. Thus, the slowest process will eventually consume all these messages. When there are no more pending messages, the process are within one round of each others, but their respective round start might still be offset by more than $3\Theta + \Delta$

Consider the case when the slowest process receives a message sent by a process already in the next round. That message will be received at most $\Theta + \Delta$ after it was sent. The catch-up mechanism of the runtime will force the slowest process to directly call the update and move to the next round, taking 2Θ. Notice that the send of the slowest will occurs $3\Theta + \Delta$ after the send of the fastest process. When the slowest process starts again receiving messages, it will be at most $3\Theta + \Delta$ behind the fastest process. Therefore, the fastest process will receives the messages from the slowest process at most $4\Theta + 2\Delta$ after is started the round. As their might be $(n-1)$ such messages, the timeout needs to be at least $(n+3)\Theta + 2\Delta$ to receive the remaining messages for the current round. Fig. 8 illustrate how the synchronizations happens.

After the system reaches synchrony. The remaining suffix π_s stays synchronous, the timeout is greater than the sum of $(n+1)\Theta + \Delta$ (timeout for the synchronous case) + $3\Theta + \Delta$ (maximal slack) - 1 (the message send by a process to itself can be processed more quickly). □

5.3 Correctness

Theorem 2. *For any* PSYNC *program $\mathcal{P}$, the transition system associated with the runtime of $\mathcal{P}$ is indistinguishable from the transition system associated with $\mathcal{P}$, i.e., $\forall\mathcal{P}.\ [\![\mathcal{P}]\!]_{rt} \trianglerighteq [\![\mathcal{P}]\!]$.*

Proof. (Sketch) Given an execution π from $[\![\mathcal{P}]\!]_{rt}$, we need to build an execution π' such that $\pi' \in [\![\mathcal{P}]\!]$ and $\pi \simeq \pi'$. We recall that the local variables V declared in $\mathcal{P}$ are modified only by the `update` operation, which has the same semantics in both $[\![\mathcal{P}]\!]$ and $[\![\mathcal{P}]\!]_{rt}$. Therefore, in order to build π' we need to show that (1) we can associate to every round r of π an execution of the same round under the PSYNC semantics, with the same output for the send operation and the same input for the `update` operation and (2) this sequence of PSYNC rounds satisfies the environment assumptions.

The execution π' is build from π by eliminating all receive steps, and shifting `send` and `update` operations to the left and right such that they execute in lockstep. Shifting the `send` and `update` of a process preserves the order of the operations within that process. For each round r the environment defines $\mathsf{HO}(p)$ to be the identity of the senders of the set of messages delivered to p tagged by round r. Without loss of generality, in $[\![\mathcal{P}]\!]_{rt}$ we assume that at each round every process sends a message to every other process. If the algorithm sends less messages we can introduce additional *ghost* messages for the proof purpose.

By Lemma 2, $\pi = \pi_a\pi_s$ where the second part is synchronous. Due to the time invariance of the environment assumptions and that π_a is finite, we can find a corresponding π'_a, a prefix of π', where the same messages are delivered. In π_s all the messages are either delivered or dropped so π_s respects the environment assumptions and the corresponding π'_s, with $\pi' = \pi'_a\pi'_s$, that also satisfies them.

Finally, the crashed processes, represented by $\perp$ in π, can be matched to any lockstep execution where the corresponding processes do not appear in the HO set of any process after the crash.

The execution π is indistinguishable from π' because (1) for every round the state preceding the `send`, resp. `update`, operations is the same in π and π'; (2) for every round the `update` operations have the same inputs in both π and π'; (3) *receive* steps in the runtime trace correspond to stuttering in the lockstep trace. □

Corollary 1. *For any* PSYNC *program $\mathcal{P}$, $[\![\mathcal{P}]\!]_{rt} \sqsubseteq [\![\mathcal{P}]\!]$.*

6. Verification

In this section we identify the class of specifications such that if a PSYNC program satisfies the specification then its runtime system satisfies it as well. We underline the advantages of PSYNC for automated verification. Finally, we present a deductive verification engine for PSYNC.

6.1 From verified PSYNC to verified runtime executions

We consider specifications given as sets of sequences of program states, or sets of runs. A specification *Spec* is a set of sequence of states in $\Sigma = [P \to V_p \to D]$, where $V_{Spec} = \bigcup_{p\in P} V_p$. The specification does not talk about the HO-sets. A spec to be applicable to a PSYNC program with the set of variables V, if

$V_{Spec} \subseteq V$. The projection of states, runs on the variables V_{Spec} is denoted by $\lfloor_{V_{Spec}}$. The projection on labels uses the same notation.

For example, consensus is defined by the conjunction of four properties: (1) Agreement, all processes decide on the same value; (2) Validity, the decision is the initial value of a process; (3) Irrevocability, a process cannot change its decision; (4) Termination, all correct processes eventually decide. These properties correspond to the following set of runs, denoted $Consensus$:

$$
\begin{aligned}
&*s_0 s_1 s_2 \ldots \in Consensus \Leftrightarrow \\
&\quad \exists q.\ \forall p.\ \forall i.\ s_i(p).\texttt{decided} \Rightarrow s_i(p).\texttt{decision} = s_0(q).\texttt{x} \\
&\wedge\ \forall p, i.\ s_i(p).\texttt{decided} \Rightarrow s_{i+1}(p).\texttt{decided} \\
&\wedge\ \forall p, i.\ s_i(p).\texttt{decided} \Rightarrow s_i(p).\texttt{decision} = s_{i+1}(p).\texttt{decision} \\
&\wedge\ \forall p.\ \exists i.\ s_i(p).\texttt{decided}.
\end{aligned}
$$

The program $\mathcal{P}$ satisfies the specification $Spec$ if all the runs of $\mathcal{P}$ are included in $Spec$, i.e., $\mathsf{Runs}(\mathcal{P})\lfloor_{V_{Spec}} \subseteq Spec$.

In the following we identify the class of specifications such that are preserved by the runtime. First, we build a transition system $TS(Spec)$ such that $\mathsf{Runs}(TS(Spec))\lfloor_{V_{Spec}} = Spec$. Then, we identify the condition under which the runtime executions of a program $\mathcal{P}$ satisfies $Spec$, if $\mathcal{P}$ satisfies $Spec$.

The modeling of crashes in the HO-model means that the specification applies only to correct processes, e.g., showing that every process decides means that every correct process decides in the runtime executions. For clients interacting with a PSYNC program, we assume that the failure of a process carries over to the client.

Transition system associated with a specification Let $Spec$ be a set of sequences of states in Σ. We build $TS(Spec) = (P, V, A, s_0, T)$ the transition systems associated with $Spec$, where P contains processes in Σ, $V_{Spec} \subseteq V$, $A = \emptyset$, $s_0 = *$. T enumerates the runs in $Spec$ such that $\mathsf{Runs}(TS(Spec))\lfloor_{V_{Spec}} = Spec$.

To relate the specification of $\mathcal{P}$ to its clients we need to add the interface labels to the specification. We relate the changes of local variables of a process to the observable labels of that process.

Definition 9 (Input-Output Interface)**.** *Let $I = \cup_{p \in P} I_p$ and f be a mapping from I to sets of states over the variables $W = \cup_{p \in P} W_p$. The pair (I, f) is an* input-output interface *of system TS if, for all p, $I_p \subseteq A_p$, $W_p \subseteq V_p$, and, for any $a \in I_p$ and $(s, B, s') \in T$, $a \in B$ iff $s(p)\lfloor_{W_p} \notin f(a) \wedge s'(p)\lfloor_{W_p} \in f(a)$.*

The input-output interface of *LastVoting* w.r.t *Consensus*, called (I_C, f_C), associates $\texttt{init}_p(v_p)$ with the first initialized state when $s(p).\texttt{x} = v$ and $\texttt{out}_p(v_p)$ when $\texttt{decided}$ is set to true and $\texttt{decision}$ is v for every $p \in P$. The mapping f_C relates $(\texttt{init}_p(v)$ to $\{s \mid s(p).\texttt{x} = v \wedge s(p).\texttt{ts} = -1\}$ and $\texttt{out}_p(v)$ to $\{s \mid s(p).\texttt{decided} \wedge s(p).\texttt{decision} = v\})$.

A PSYNC program might require the addition of variables to ensure that input-output interfaces exists. For instance, the $\texttt{decision}$ variable in *LastVoting* changes values when the $\texttt{out}$ event occurs.

Given a specification $Spec$ and an input-output interface (I, f), $TS(Spec, f)$ is the transition system obtained from $TS(Spec)$ by setting $A = I$ and changing T to add labels according to Def. 9.

If an interface is compatible with a program and a specification then it relates the program's executions to the executions of the transition system obtained from the specification.

Proposition 1. *For any* PSYNC *program $\mathcal{P}$, a specification $Spec$, and an input-output interface (I, f) for $\mathcal{P}$ and $Spec$:*

$$\mathsf{Runs}(\mathcal{P})\lfloor_{V_{Spec}} \subseteq Spec \Rightarrow [\![\mathcal{P}]\!]\lfloor_{V_{Spec}, I} \subseteq [\![TS(Spec, f)]\!]\lfloor_{V_{Spec}, I}$$

To relate a specification to the runtime executions, we need to account for indistinguishability. Given a transition system TS we denote by $Closure_{\simeq}(TS)$ its closure w.r.t. the indistinguishability relation, i.e., $[\![Closure_{\simeq}(TS)]\!] = \{\pi \mid \exists \pi' \in [\![TS]\!].\ \pi \simeq \pi'\}$.

Theorem 3. *For any* PSYNC *program $\mathcal{P}$ with an input-output interface (I, f), and specification $Spec$, if $\mathcal{P}$ satisfies $Spec$ and $TS(Spec)$ is closed under indistinguishability, then*

$$[\![\mathcal{P}]\!]_{rt} \trianglerighteq [\![TS(Spec, f)]\!] \text{ and } [\![\mathcal{P}]\!]_{rt} \sqsubseteq_I TS(Spec, f).$$

Proof. Let $\pi \in [\![\mathcal{P}]\!]_{rt}$ be an execution of the runtime of $\mathcal{P}$. Th. 2 implies that there exists $\pi' \in [\![\mathcal{P}]\!]$ s.t. $\pi \simeq \pi'$. Moreover, since V_{Spec} is included in the set of program variables in $\mathcal{P}$ and the interface I is common for $\mathcal{P}$ and its runtime, (and indistinguishability between the runtime and PSYNC holds w.r.t. all the variables in $\mathcal{P}$ and all common labels), it implies that $\pi\lfloor_{V_{Spec}, I} \simeq \pi'\lfloor_{V_{Spec}, I}$.

Since $\mathcal{P}$ satisfies the specification, from Prop. 1 it follows that $\pi' \lfloor_{V_{Spec}, I} \in [\![TS(Spec, f)]\!]\lfloor_I$. Finally, since $[\![TS(Spec, f)]\!]$ is closed under indistinguishability $\pi \lfloor_{V_{Spec}, I} \in [\![TS(Spec, f)]\!]\lfloor_I$. Therefore, $[\![\mathcal{P}]\!]_{rt} \trianglerighteq [\![TS(Spec, f)]\!]$ and, using Th. 1, it follows that $[\![\mathcal{P}]\!]_{rt} \sqsubseteq_I TS(Spec, f)$. □

Proposition 2. *Consensus is closed under indistinguishability and if a program $\mathcal{P}$ satisfies Consensus then $[\![\mathcal{P}]\!]_{rt} \sqsubseteq_{I_C} [\![\mathcal{P}]\!] \sqsubseteq_{I_C} TS(Consensus, f_C)$.*

Roughly, consensus is closed under indistinguishability because it does not impose an order on the updates performed on different processes: processes can decide on a value in any order and at any time. The same reasoning also holds for other agreement specifications, like the k-set agreement [26], or the lattice agreement [33].

6.2 Benefits of PSYNC for verification

Distributed algorithms are challenging to verify because of several sources of unboundedness. Messages come from unbounded domains, the number of processes is a parameter, and channels may also be unbounded. Using communication-closed rounds and a lockstep semantics helps mitigate or avoid these challenges.

Model checking Model checking techniques are based on algorithms that explore the system's reachable states. It requires a fixed number of finite state processes. With an asynchronous semantics, a model checker explores all the possible interleavings of processes transitions and suffers from combinatorial explosion. In the lockstep semantics of PSYNC all the processes take a step at the same time removing the interleavings. Another difficulty comes from the communication channels. Unbounded FIFO channels causes undecidability even for two processes [17]. Making the channels lossy [3] and fixing the number of processes makes the problem non-primitive recursive [64]. Weaker channel models are usually at least EXPSPACE-hard for verification. Communication-closed rounds sidestep this difficulty.

Deductive verification Deductive verification relies on user provided inductive invariants and ranking functions. The invariants describe an over-approximation of the set of reachable states which is inductive w.r.t. the program transitions. Ranking functions show progress toward satisfying the program goals. However, finding these annotations is not easy even for an expert. Automated techniques, such as static analysis, are far from being able to generate these annotations automatically for our targeted class of systems.

The lockstep semantics leads to much simpler invariants, because they are required to describe the set of reachable states only at the boundaries between rounds.

In the literature, the HO-model has been shown to be suited for verification using bounded state-space exploration [22, 66–68] and interactive theorem provers [23, 25, 27, 54].

6.3 A verifier for PSYNC

We consider specifications that include both safety and eventuality properties and are formally defined in LTL with state properties in the logic $\mathbb{CL}$ [30]. The verifier inputs are the specification and a PSYNC program annotated with inductive invariant candidates. The verifier checks the validity of the invariants and that they imply the specification by generating verification conditions that can be discharged using an SMT solver.

Expressiveness $\mathbb{CL}$ is a first-order logic over sets of program states. The variables are interpreted over the different types declared in the program. The value of the program variable `x` of type `T` of a process p is denoted in the logic by the term $\texttt{x}(p)$, where `x` is a function of type $P \to \texttt{T}$ with P is the type of processes. To characterize global states, $\mathbb{CL}$ uses universal quantification over variables of type P, set comprehensions, and cardinality constraints.

The programmer provides the inductive invariant candidates and the pre/post condition of the `send`, `update` functions. Typically the correctness argument for consensus solving algorithms, which an invariant must capture, is centred around the existence of *majority of processes* that support a decision. For example, the formula $\exists v.\ |\{p \mid \texttt{x}(p) = v\}| > \texttt{n}/2$ defines a majority ($> \texttt{n}/2$) of processes that agree on value v using a comprehension, where `x` is the function symbol associated with the local variable `x`.

The inductive invariant that shows agreement in *LastVoting* is

$$\begin{array}{lll} & \forall p. & \neg\texttt{decided}(p) \wedge \neg\texttt{ready}(p) \\ \vee & \exists v, t, A. & A = \{p \mid \texttt{ts}(p) \geq t\} \ \wedge \ t \leq r/4 \\ & \wedge & |A| > n/2 \ \wedge \ \forall p.\, p \in A \Rightarrow \texttt{x}(p) = v \\ & \wedge & \forall p.\, \texttt{decided}(p) \Rightarrow \texttt{decision}(p) = v \\ & \wedge & \forall p.\, \texttt{commit}(p) \vee \texttt{ready}(p) \Rightarrow \texttt{vote}(p) = v \\ & \wedge & \forall p.\, \texttt{ts}(p) = r/4 \Rightarrow \texttt{commit}(\texttt{coord}(r/4)). \end{array}$$

The invariant is a case split characterizing the states in which processes can safely decide. A process decides when there is a majority of processes agreeing on a proposal with timestamps more recent than t. The additional clauses are required to make the invariant inductive and to relate it to the specification. For instance, if a process is `ready` then its vote is v and it agrees with the majority. Also any process that has decided its decision is v.

Methodology To prove safety properties we implement a standard verification conditions generator for PSYNC programs. For the round R, the generator builds a $\mathbb{CL}$ formula corresponding to the transitions relation as follows. Let s, s' be the primed and unprimed function used to represent the global state of the system. The transition relation associated with a local `send`, resp. `update`, of a round R is $\texttt{send}_R(s(p), m)$ resp. $\texttt{update}_R(m, s(p), s'(p))$.

Then the transition relation of R, $TR_R(s, s')$ is

$$\begin{array}{lll} & \forall p. & \texttt{send}_R(s(p), m_s(p)) \\ \wedge & \forall p, q, t. & (t, q) \in m_u(p) \Leftrightarrow (t, p) \in m_s(q) \wedge q \in \mathsf{HO}(p) \\ \wedge & \forall p. & \texttt{update}_R(m_u(p), s(p), s'(p)) \ \wedge \ r' = r + 1. \end{array}$$

Safety We generate verification conditions that imply partial correctness: (1) the invariant contains the initial state ($TR_{\texttt{init}}(s) \Rightarrow Inv(s)$), (2) for any round R the invariant is inductive ($Inv(s) \wedge TR_R(s, s') \Rightarrow Inv(s')$), (3) the safety specification φ is implied by the invariant ($Inv(s) \Rightarrow \varphi(s)$).

Eventuality We also prove eventuality properties, such as, every process eventuality decides in *LastVoting*. Showing these properties typically requires ranking functions. However, in many cases we can simplify the proof. We show that there exists a fixed number of good rounds, i.e., rounds when the environment assumptions hold, such that after the execution of the last good round the program reaches a set of good states, e.g., processes have decided. To prove that the program makes progress after each good round the user provides additional invariants, expressing how a good round strengthens the safety invariant. For instance, in the *LastVoting* the program makes a decision if the formula (1) holds during one complete phase of the algorithms. An intermediate invariants specify that between round `Collect` and `Candidate` the formula $\texttt{commit}(\texttt{coord}(r))$ holds on top of the safety invariant.

Algorithm implemented in PSYNC	LOC	Use rounds	Async.
Last voting [24]	89	✓	✓
One third rule [24]	50	✓	✓
Flood min consensus [51]	22	✓	×
Ben-Or randomized consensus [13]	58	✓	✓
k-set agreement [26]	39	✓	✓
k-set agreement early stopping [61]	30	✓	×
Lattice agreement [33]	30	×	✓
ϵ-agreement [46]	49	✓	✓
Two phases commit [38]	53	✓	×
Eager reliable broadcast [19]	27	×	×

Table 1: Fault-tolerant algorithms implemented in PSYNC

7. Evaluation

We have implemented PSYNC as an embedding in the SCALA programming language. The runtime of PSYNC is built on top of the NETTY [2] framework. For the transport layer, we use UDP. The serialization of messages uses the pickling library [56].

The set of replicas executing a PSYNC program is specified in a configuration file. The runtime manages the interface between a PSYNC program $\mathcal{P}$ and the client application, and also the communication between the different replicas running $\mathcal{P}$.

The execution of $\mathcal{P}$ is launched from a client application, using $\mathcal{P}$'s interface. More specifically, the interface contains callback methods provided by the application. Regarding termination, many consensus algorithms presented in the literature assume that processes continue executing the algorithm after a decision is taken, because not all processes decide simultaneously. For example, in a case of a network partition, some processes learn the decision value much later. To safely free the memory allocated by the runtime when a process decides, each replica stores only the decision value in a log. In our experiments we implement a key-value store using *LastVoting* iteratively. On each replica we terminate an execution of *LastVoting* after a process decides and keep a log of the most recent decisions. Replicas which have terminated executing *LastVoting* detect messages from the late replicas and send them the decision.

To achieve the good performances, the timeout is an important parameter determined empirically. Δ can easily be measured, i.e., latency and bandwidth. Θ is harder to measure but can easily be over-approximated. To decrease the reliance on an accurate timeout, we implemented several optimizations that allows the runtime to progress before the timeout occurs.

7.1 Implementing Algorithms in PSYNC

PSYNC can implement a wide variety of fault-tolerant distributed algorithms. Table 1 lists several implementations in PSYNC of algorithms that solve different agreement problems. For each algorithm, we indicate if it is designed for a synchronous or asynchronous network and if the presentation uses some form of rounds. Many asynchronous algorithms are tagging messages with information that implicitly structures programs in rounds (not necessarily executing in lockstep). However, even when the original algorithm presentation is event-driven, they can be encoded in PSYNC.

The first three algorithms focus on the traditional consensus problem, the others are weaker agreement problem. Ben-Or algorithm [13] solves binary consensus and almost surely terminates. The k-set agreement [26] is a weaker version of consensus that allows processes to decide on k-different values. The generalized

Paxos implemented in	LOC	Executable	Verification
PSYNC	89	✓	semi-automated
DistAlgo	43	✓	×
Distal	157	✓	×
Overlog	107	✓	×
TLA+	53	×	mechanized
IO Automata	142	×	mechanized
EventML	1729N	✓	mechanized
Verdi (Raft algorithm)	520	✓	mechanized
Bloom	224	✓	×

Table 2: Comparison of the code size and verification of Paxos in different languages. For EventML, Schiper et. al. [63] report the number of AST nodes. The sources locations are in Appendix C.

lattice agreement [33] asks processes to choose values in a lattice, such that these values form a chain. ϵ-agreement is a form of consensus over $\mathbb{R}$ in which all the decision values lie in an interval of size ϵ. Two phases commit is a degenerate version of the binary consensus where the decision value $true$ is allowed only if all processes propose $true$. Reliable broadcast guarantees that if a correct process delivers a value, then all correct processes deliver that value.

We compare PSYNC against other high-level languages for distributed algorithms. Table 2 shows a comparison between different implementations of Paxos in different programming and specification languages. For PSYNC we used *LastVoting*. For each implementation we count the number of lines of code without comments or blank lines. Also we focus on the algorithm itself and remove boilerplates like include statements.

We compare against the following languages: DistAlgo [50] is a programming language for distributed algorithm that uses incrementalization to compile a high-level specification into Python code. Distal [16] is designed to express fault-tolerant distributed algorithms in a pseudo-code-like manner. It is built as a library on top of Scala. Bloom [7] is a programming language based on a monotonic logic for building consistent distributed systems. Overlog [6] is a logic programming language for distributed systems inspired by Datalog. EventML [55] is a programming and verification language for distributed algorithms connected to the Nuprl interactive theorem prover [5]. Verdi [70] is a Coq framework for implementing and proving distributed systems correct. TLA+ [48] is a logic-based specification language designed to describe concurrent and distributed systems. IO Automata [52] is a specification language with automata theoretic foundations to describe asynchronous concurrent and distributed systems. Except for TLA+ which can encode both synchronous and asynchronous programs, and Verdi that starts with a synchronous model and transforms it into an asynchronous one, the other languages have an asynchronous semantics. Currently only mechanized proofs in Nuprl or Coq exist for Paxos, when implemented in EventML or Verdi.

Limitations of the model PSYNC keeps an order between messages only if they are sent in different rounds. It is oblivious to the order in which messages arrive in one round. As a consequence, one cannot implement the runtime system of PSYNC in PSYNC itself. Also, for the moment we don't have an efficient way of composing PSYNC programs to create other programs. The composing round based models is a problem that currently receives attention.

7.2 Comparing PSYNC to existing Paxos implementations

We evaluate our PSYNC implementation of Paxos (*LastVoting* from Fig. 1) and compare it to existing Paxos implementations. We use *LastVoting* to order write requests in a simple key-value store. The submitted requests are collected into batches of about 300 requests, then *LastVoting* is used to make all replicas agrees on the next set of writes. Adding batching to *LastVoting* requires only changing the type of the messages sent in each round, i.e., modifying only a few lines of code (<10) because batching does not interfere with the control structure of the algorithm.

Implementation	Source	Year	Throughput ×1000 req/s
Last Voting (Batching)		2015[3]	170
Egalitarian Paxos	[59]	2013	450
Paxos in Distal	[16]	2013	150
JPaxos / SPaxos	[15]	2012	75 / 300
Paxos for system builder	[8]	2008	40

Table 3: Performance of Paxos implementations with 3 replicas

Table 3 shows throughput numbers for the different implementations of Paxos we considered. All the algorithms try to maximize the throughput when dealing with requests of small sizes. However, the exact settings is sightly different for every experiment. Due to the difficulty of replicating published results, we report the published numbers. We run PSYNC on three servers with Intel Xeon X5460 cpu and 8 GB ram [3], running Linux 2.6.32 and the JRE 1.8. We also ran experiments incorporating crashes. The throughput of the system roughly halves after one crash. The point of this comparison is to show that the overhead of the runtime to implement the round structure is acceptable and does not preclude PSYNC adoption. We believe that the benefits of PSYNC, i.e., an intuitive semantics, simple control structures, and the ability to use automated verification tools, make it a compelling language.

PSYNC and Distal both are based on SCALA. JPaxos and SPaxos [15] are written in Java. JPaxos is Java implementation of Paxos and SPaxos is an improved algorithm to achieve higher-throughput when the coordinator is CPU bound. Egalitarian Paxos [59] is implemented in Go and improves over Paxos by processing independent requests in parallel. Paxos for system builder [8] is an implementation of Paxos in C. To achieve high throughput, all the implementations use batching.

7.3 Verification in PSYNC

We implemented a verification conditions generator for PSYNC, based on the logic $\mathbb{CL}$. To compute the verification conditions the tool computes, at compile time, (1) the transition relation of the program and (2) transforms the program assertions given in the SCALA language in $\mathbb{CL}$ formulas.

We implemented the semi-decision procedure for $\mathbb{CL}$ [30] on top of the SMT solver Z3 [60] and using the VC generator we verified the PSYNC programs One third rule [24] and *LastVoting*. Their specification and invariants are provided by the user. We used the invariants from [30]. For the One Third Rule, we need for 4 invariants (23 LOCs), 27 VCs are generated, solved in 5s. For the *LastVoting*, we need for 8 invariants (35 LOCs), 45 VCs are generated, solved in 16s.

The verification of programs solving weaker agreement problem requires reasoning about sets of data. For example, k-set consensus needs to reason about the cardinality of the set of decision values. $\mathbb{CL}$ supports only reasoning about sets of processes. We are working on extending the scope of our implementation to verify a larger class of examples.

8. Related Work

In this section we compare PSYNC to the related work from a programming language and verification perspective.

Formalizations of distributed algorithms Distributed algorithms are typically defined in English or pseudo-code [39] using different

[3] We use 5 years old machines which makes the comparison with older results relevant

computational models. Synchronous and asynchronous models are the most frequent ones. Synchrony allows solving a larger class of problems, while asynchrony is close to the network behavior. Finding an uniform model is still an open problem in the distributed algorithms community. Multiple models that abstract uniformly faults and (a)synchrony have been introduced [4, 14, 24, 31, 37, 62, 69]. We have chosen the HO-model [24] because of its simplicity. It handles asynchrony, host and network failures uniformly.

In the classic setting, distributed systems are formalized using the π-calculus [57, 58], CSP [42], and I/O-automata [52]. These formalisms are used to give a formal semantics to message-passing systems and to analyze them, but not as programming languages.

The Actor model [41] is probably the most successful high-level programming abstraction for message-passing systems. Actors are either built-in languages, e.g., Erlang [10], or supported through libraries, e.g., Scala [40]. Erlang via the OTP library [65] has support to handle faults in distributed systems. Faults are handled using a supervision hierarchy: when a replica fails its superior in the hierarchy is notified and takes action. PSYNC allows reasoning about faults when processes are not organized in a strict hierarchy.

Several domain specific languages for distributed algorithms have been developed [6, 7, 11, 16, 45, 50, 55]. Languages like Meld [11], Overlog [6], and Bloom [7], which are based on Datalog, do not have a formal operational semantics, and do not support automated verification. The closest domain specific languages to PSYNC are Mace [45], DistAlgo [50] and Distal [16]. However, they all have an asynchronous semantics and lack high-level programming construct to reason explicitly about faults.

Verification The verification of parametric systems is in general undecidable [9]. Therefore, mainly bug finding tools are developed. They are based on state-space exploration up to a bounded, generally small, number of processes or messages [29, 36, 45, 66, 67], or uses specialized abstractions [43]. Static analysis techniques [1] are used to prove only simple properties, such as type errors or dead code detection, not for complex functional properties.

Recently a few programming languages were designed with build-in support for verification. Mace [45] has an integrated model checker which cannot prove total functional correctness. The most successful programming languages from a verification perspective are EventML [55] and Verdi [70]. EventML is a functional programming language and Verdi is a Coq framework for implementing and proving correct distributed algorithms. Verdi starts with a synchronous implementation and progressively transforms it using refinement into an asynchronous fault-tolerant one. The correctness of the implemented programs is mechanically proved using theorem provers: EventML uses Nuprl and Verdi uses Coq. These two languages have a small trusted base but reduced performances. For example, the throughputs of PSYNC implementations are several orders of magnitude faster than corresponding ones in Verdi. Moreover PSYNC's verifier is designed for automated verification, currently based on SMT solvers.

In the algorithms community specification languages like +Cal [49] and TLA+ [48] are used to write formal specification of distributed algorithm and to model check or to mechanically prove them. However, these specifications are not executable.

Finally, the exploration of synchronous behavior of asynchronous systems for verification purposes has been investigated in [12, 28]. Our approach starts with a (partially) synchronous abstraction rather than retrofitting synchrony in existing asynchronous systems. The approach in [12, 28] makes the verification more complex without offering any guarantees about its applicability.

9. Conclusion

We have presented PSYNC a domain specific language for fault-tolerant systems that strikes a balance between high-level constructs, performance, and automated verification. PSYNC offers a simple lockstep semantics that is indistinguishable from its runtime asynchronous executions. We have implemented a prototype runtime for PSYNC for partial synchronous networks and shown that it performs within a constant factor from highly optimized low-level implementations. For future work we intend to enlarge the application domain of PSYNC and to raise the automation level of the verification engine by developing static analysis that generate inductive invariants. We plan to generalize the runtime to cover more fault-models, such a Byzantine faults.

Acknowledgments This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM) and by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award). Yours XXX I think we should thank Josef and while at it the reviews. Josef didn't contribute to the paper but the initial discussions with him were useful.

References

[1] Dialyzer. http://www.erlang.org/doc/man/dialyzer.html.

[2] The netty project. http://netty.io/. Accessed: 2015-06-03.

[3] P. A. Abdulla, A. Collomb-Annichini, A. Bouajjani, and B. Jonsson. Using Forward Reachability Analysis for Verification of Lossy Channel Systems. *FMSD*, 25(1):39–65, 2004.

[4] Y. Afek and E. Gafni. Asynchrony from synchrony. In D. Frey, M. Raynal, S. Sarkar, R. K. Shyamasundar, and P. Sinha, editors, *Distributed Computing and Networking, 14th International Conference, ICDCN 2013, Mumbai, India, January 3-6, 2013. Proceedings*, pages 225–239, 2013.

[5] S. F. Allen, R. L. Constable, R. Eaton, C. Kreitz, and L. Lorigo. The nuprl open logical environment. In D. A. McAllester, editor, *Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, pages 170–176, 2000.

[6] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I do declare: consensus in a logic language. *Operating Systems Review*, 43 (4):25–30, 2009.

[7] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011. URL http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf.

[8] Y. Amir and J. Kirsch. Paxos for system builders: An overview. In *Workshop on Large-Scale Distributed Systems and Middleware (LADIS 2008), Yorktown, NY, September 2008*, 2008.

[9] K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[10] J. L. Armstrong. The development of erlang. In S. L. P. Jones, M. Tofte, and A. M. Berman, editors, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997.*, pages 196–203. ACM, 1997.

[11] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. Campbell. A language for large ensembles of independently executing nodes. In *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, pages 265–280, 2009.

[12] S. Basu, T. Bultan, and M. Ouederni. Synchronizability for verification of asynchronously communicating systems. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 56–71, 2012.

[13] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC*, pages 27–30. ACM, 1983.

[14] M. Biely, B. Charron-Bost, A. Gaillard, M. Hutle, A. Schiper, and J. Widder. Tolerating corrupted communication. In *PODC*, pages 244–253, 2007.

[15] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *IEEE 31st Symposium on Reliable Distributed Systems, SRDS 2012, Irvine, CA, USA, October 8-11, 2012*, pages 111–120, 2012.

[16] M. Biely, P. Delgado, Z. Milosevic, and A. Schiper. Distal: A framework for implementing fault-tolerant distributed algorithms. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*, pages 1–8, 2013.

[17] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.

[18] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1.

[19] C. Cachin, R. Guerraoui, and L. Rodrigues. Reliable broadcast. In *Introduction to Reliable and Secure Distributed Programming*, pages 73–135. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-15259-7.

[20] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[21] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-616-5.

[22] M. Chaouch-Saad, B. Charron-Bost, and S. Merz. A reduction theorem for the verification of round-based distributed algorithms. In *RP*, volume 5797 of *LNCS*, pages 93–106, 2009.

[23] B. Charron-Bost and S. Merz. Formal verification of a consensus algorithm in the heard-of model. *Int. J. Software and Informatics*, 3(2-3):273–303, 2009.

[24] B. Charron-Bost and A. Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1): 49–71, 2009.

[25] B. Charron-Bost, H. Debrat, and S. Merz. Formal verification of consensus algorithms tolerating malicious faults. In *SSS*, pages 120–134. Springer, 2011. ISBN 978-3-642-24549-7.

[26] S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132 – 158, 1993. ISSN 0890-5401.

[27] H. Debrat and S. Merz. Verifying fault-tolerant distributed algorithms in the heard-of model. *Archive of Formal Proofs*, 2012, 2012.

[28] A. Desai, P. Garg, and P. Madhusudan. Natural proofs for asynchronous programs using almost-synchronous reductions. In A. P. Black and T. D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 709–725, 2014.

[29] E. D'Osualdo, J. Kochems, and C. L. Ong. Automatic verification of erlang-style concurrency. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 454–476, 2013.

[30] C. Dragoi, T. A. Henzinger, H. Veith, J. Widder, and D. Zufferey. A logic-based framework for verifying consensus algorithms. In K. L. McMillan and X. Rival, editors, *VMCAI*, pages 161–181. Springer, 2014.

[31] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, 35(2):288–323, Apr. 1988.

[32] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Sci. Comput. Program.*, 2(3):155–173, 1982.

[33] J. M. Falerio, S. K. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani. Generalized lattice agreement. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 125–134, 2012.

[34] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010. . URL http://dx.doi.org/10.1016/j.tcs.2010.09.021.

[35] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[36] L. Fredlund and H. Svensson. Mcerlang: a model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 125–136, 2007.

[37] E. Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In B. A. Coan and Y. Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 143–152, 1998.

[38] J. Gray. Notes on data base operating systems. In R. Bayer, R. Graham, and G. Seegmüller, editors, *Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer Berlin Heidelberg, 1978. ISBN 978-3-540-08755-7.

[39] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 3540288457.

[40] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009. . URL http://dx.doi.org/10.1016/j.tcs.2008.09.019.

[41] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, August 1973*, pages 235–245, 1973.

[42] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[43] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *FMCAD*, pages 201–209, 2013.

[44] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, pages 245–256. IEEE, 2011.

[45] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: language support for building distributed systems. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 179–188, 2007.

[46] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011.

[47] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16 (2):133–169, May 1998. ISSN 0734-2071.

[48] L. Lamport. Distributed algorithms in TLA (abstract). In *PODC*, 2000.

[49] L. Lamport. The pluscal algorithm language. In *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, pages 36–60, 2009.

[50] Y. A. Liu, S. D. Stoller, B. Lin, and M. Gorbovitski. From clarity to efficiency for distributed algorithms. In G. T. Leavens and M. B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 395–410, 2012.

[51] N. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[52] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In F. B. Schneider, editor, *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, pages 137–151, 1987.

[53] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machine for wans. In R. Draves and R. van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 369–384. USENIX Association, 2008. URL http://www.usenix.org/events/osdi08/tech/full_papers/mao/mao.pdf.

[54] O. Marić, C. Sprenger, and D. Basin. Consensus refined. In J. Karlsson and Y. Amir, editors, *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015.

[55] V. R. Mark Bickford, Robert L. Constable. Logic of events, a framework to reason about distributed systems. In *2012 Languages for Distributed Algorithms Workshop*, Philadelphia, PA, 2012. URL http://www.nuprl.org/documents/Bickford/LOE-LADA2012.html.

[56] H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *OOPSLA*, pages 183–202, 2013.

[57] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[58] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

[59] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8.

[60] L. Moura and N. Bjorner. Z3: An efficient SMT solver. In C. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78799-0.

[61] P. Parvedy, M. Raynal, and C. Travers. Early-stopping k-set agreement in synchronous systems prone to any number of process crashes. In V. Malyshkin, editor, *Parallel Computing Technologies*, volume 3606 of *Lecture Notes in Computer Science*, pages 49–58. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-28126-9.

[62] N. Santoro and P. Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theor. Comput. Sci.*, 384(2-3):232–249, 2007.

[63] N. Schiper, V. Rahli, R. van Renesse, M. Bickford, and R. L. Constable. Developing correctly replicated databases using formal tools. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 395–406, 2014.

[64] P. Schnoebelen. Revisiting Ackermann-Hardness for Lossy Counter Machines and Reset Petri Nets. In *MFCS*, pages 616–628, 2010.

[65] S. Torstendahl. Open telecom platform. *Ericsson Review*, 1, 1997.

[66] T. Tsuchiya and A. Schiper. Model checking of consensus algorithms. In *SRDS*, pages 137–148, 2007.

[67] T. Tsuchiya and A. Schiper. Using bounded model checking to verify consensus algorithms. In *DISC*, pages 466–480, 2008.

[68] T. Tsuchiya and A. Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23(5-6):341–358, 2011.

[69] J. Widder and U. Schmid. The Theta-Model: Achieving synchrony without clocks. *Distributed Computing*, 22(1):29–47, Apr. 2009.

[70] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In D. Grove and S. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 357–368. ACM, 2015.

A. Applicability to network assumptions

In the above, we focus on the case of partially synchronous system where crashed processes do not recover. However, PSYNC is not restricted this assumption and the runtime can be adapted for a wider variety of systems. We now give an overview of how to implement a round structure under different assumptions.

Alternative definition of partial synchrony. The runtime algorithm also works with the definition of partial synchrony where the bounds are not known. In this setting, the runtime must use incremental timeout, e.g., on line 19 in Fig. 6. A more realistic model allows the alternation of good and bad periods. Our system can works in this context. The challenges are discussed by Dwork et.al. in [31]

Stronger safety assumptions. The HO-model allow not only liveness assumptions, but also safety assumptions, which must always hold. Asynchrony corresponds to the safety assumptions $true$.

For safety assumptions that are stronger than asynchrony, another runtime algorithms may by needed. If the network provides timing guarantees those can be used. If the safety assumptions are locally testable, then the algorithm in Figure 6 can be modified so the non-satisfaction of the test can override the timeout until enough messages are received. Also the reception policy may change to processing messages with the lowest round numbers first.

Crash-recovery. Crash-recovery is also possible. To allow recovery the runtime must store in stable memory (hard disk) the state of the system before sending messages. Upon recovery, the system resumes from the last state in the stable memory.

Byzantine failures. Byzantine failure requires a more complex algorithm to implement the round structure. First, the messages needs to be signed (HMAC) to avoid one process trying to impersonate another. Also the moving to the next round requires receiving enough messages ($2/3$ majority). Otherwise, a Byzantine process can send messages with higher and higher round number and keep the system in an eternal catching-up game.

B. Additional definitions

Definition 10 (Distributed client)**.** *Let $TS_i = (\{p_i\}, V_i, A_i, s_0^i, T_i)$ be the transition system associated with a client process, with $A_i \cap A_j = \emptyset$ for all $1 \leq i \neq q \leq n$. Formally, the transitions system associated with the client is $C_{TS} = (P, V, A, s_0, T)$, where $P = \{p_1, p_2, \dots p_n\}$, $V = \biguplus_i V_i$, $A = \bigcup_i A_i$, $s_0 = (s_0^1, \dots, s_0^n)$, and $T \subseteq \Sigma \times A \times \Sigma$, with $\Sigma = [P \to V \to \mathcal{D}]$, such that $\Sigma(p_i) \in [V_i \to \mathcal{D}]$, and $(s, B, s') \in T$ iff for every $b \in B \cap A_i$ $(s(p_i), b, s'(p_i)) \in T_i$ and each processes takes at most one transition.*

Definition 11 (Synchronized product)**.** *Let TS_1 and TS_2 be two labeled transition systems such that $I \subseteq A_2$ is an interface of TS_2 and $I \subseteq A_1$. We defined the* synchronized product of TS_1 and TS_2 with respect to I*, denoted $TS_1 \times_I TS_2$, as the asynchronous product of transition systems where the transition labeled in I are synchronized. Formally, $TS_1 \times_I TS_2 = (P, V, s_0, T, A)$, where $P = P_1 \cup P_2$ $V = V_1 \uplus V_2$, $s_0 = (s_0^1, s_0^2)$, $\Sigma = \Sigma_1 \times \Sigma_2$, $A = A_1 \cup A_2$, and for any $s, s' \in \Sigma_1$, $t, t' \in \Sigma_2$, for any $a \in A_1 \setminus I$, if $(s, a, s') \in T_1$ then $((s, t), a, (s', t)) \in T$, for any $a \in A_2 \setminus I$, if $(t, a, t') \in T_2$ then $((s, t), a, (s, t')) \in T$, and for any $a \in I$, if $(s, a, s') \in T_1$ and $(t, a, t') \in T_2$ then $((s, t)a(s', t')) \in T$.*

C. Sources of for the programs size comparaison

DistAlgo https://github.com/DistAlgo/distalgo/blob/master/examples/lapaxos/orig.da

Distal https://github.com/distal/distal-examples/blob/master/src/main/scala/ch/epfl/lsr/paxos/parliament.scala

Bloom https://github.com/bloom-lang/bud-sandbox/tree/master/paxos

Overlog https://bitbucket.org/neilconway/overlog-paxos/src/tip/src/olg/core/election.olg

IO Automoata in the IO-Automoata toolkit http://groups.csail.mit.edu/tds/ioa/

TLA+ https://github.com/fintler/tlaplus/blob/master/examples/Paxos/Paxos.tla

Verdi we took the number reported in [70].

EventML we took the number reported in [63].

These references were accessed on 21st of March 2015.